AD-779 883

A RATIONALE AND DESCRIPTION OF THE BASIC INSTRUCTIONAL PROGRAM

STANFORD UNIVERSITY

PREPARED FOR
OFFICE OF NAVAL RESEARCH
ADVANCED RESEARCH PROJECTS AGENCY

22 APRIL 1974

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>Technical Report No. 5 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)*<br>A Rationale and Description of the BASIC Instructional Program | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical Report |
| | | 6. PERFORMING ORG. REPORT NUMBER<br>Technical Report No. 228 |
| 7. AUTHOR(s)<br>Avron Barr, Marian Beard, and Richard C. Atkinson | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-67-A-0012-0054 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Institute for Mathematical Studies in the Social Sciences - Stanford University<br>Stanford, California 94305 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>61153N<br>RR 042-0; RR 042-0-0<br>NR 154-326 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Personnel and Training Research Programs<br>Office of Naval Research (Code 458)<br>Arlington, VA 22217 | | 12. REPORT DATE<br>April 22, 1974 |
| | | 13. NUMBER OF PAGES<br>55 |
| 14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* | | 15. SECURITY CLASS. *(of this report)*<br>Unclassified |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

BASIC, Computer-Assisted Instruction (CAI), computer programming, computer science education, instruction control strategy, tutorial CAI

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A course in computer programming is being developed as a vehicle for research in tutorial modes of computer-assisted instruction. Methods for monitoring and aiding the student as he works on interesting programming problems are employed. The problems are individually selected via an optimization scheme based on a model of the student's ability and difficulties.

DD FORM 1473 1 JAN 73    EDITION OF 1 NOV 65 IS OBSOLETE<br>S/N 0102-014-6601

UNCLASSIFIED

After a brief overview of work done at Stanford in tutorial CAI and the teaching of procedural skills, the functional elements of the BASIC Instructional Program, its BASIC interpreter, curriculum, solution analysis, and interactive assistance during programming, are described.

At BIP's core is an information network which embodies the interrelations of the concepts, skills, problems, remedial lessons, hints, BASIC commands, and manual references. With the data stored in the student history, the network enables BIP to model the student's state of knowledge, and to make problem selections with some relevance. The sophistication of these modelling techniques are the main thrust of our research.

# TECHNICAL REPORTS

## PSYCHOLOGY SERIES

### INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES

(Place of publication shown in parentheses; if published title is different from title of Technical Report,
this is also shown in parentheses.)

125   W. K. Estes. Reinforcement in human learning. December 20, 1967. (In J. Tapp (Ed.), Reinforcement and behavior. New York: Academic Press, 1969. Pp. 63-94.)

126   G. L. Wolford, D. L. Wessel, and W. K. Estes. Further evidence concerning scanning and sampling assumptions of visual detection models. January 31, 1968. (Perception and Psychophysics, 1968, 3, 439-444.)

127   R. C. Atkinson and R. M. Shiffrin. Some speculations on storage and retrieval processes in long-term memory. February 2, 1968. (Psychological Review, 1969, 76, 179-193.)

128   J. Holmgren. Visual detection with imperfect recognition. March 29, 1968. (Perception and Psychophysics, 1968, 4(4),        .)

129   L. B. Mlodnosky. The Frostig and the Bender Gestalt as predictors of reading achievement. April 12, 1968.

130   P. Suppes. Some theoretical models for mathematics learning. April 15, 1968. (Journal of Research and Development in Education, 1967, 1, 5-22.)

131   G. M. Olson. Learning and retention in a continuous recognition task. May 15, 1968. (Journal of Experimental Psychology, 1969, 81, 381-384.)

132   R. N. Hartley. An investigation of list types and cues to facilitate initial reading vocabulary acquisition. May 29, 1968. (Psychonomic Science, 1968, 12(b), 251-252; Effects of list types and cues on the learning of word lists. Reading Research Quarterly, 1970, 6(1), 97-121.)

133   P. Suppes. Stimulus-response theory of finite automata. June 19, 1968. (Journal of Mathematical Psychology, 1969, 6, 327-355.)

134   N. Moler and P. Suppes. Quantifier-free axioms for constructive plane geometry. June 20, 1968. (Compositio Mathematica, 1968, 20, 143-152.)

135   W. K. Estes and D. P. Horst. Latency as a function of number of response alternatives in paired-associate learning. July 1, 1968.

136   M. Schlag-Rey and P. Suppes. High-order dimensions in concept identification. July 2, 1968. (Psychometric Science, 1968, 11, 141-142.)

137   R. M. Shiffrin. Search and retrieval processes in long-term memory. August 15, 1968.

138   R. D. Freund, G. R. Loftus, and R. C. Atkinson. Applications of multiprocess models for memory to continuous recognition tasks. December 18, 1968. (Journal of Mathematical Psychology, 1969, 6, 576-594.)

139   R. C. Atkinson. Information delay in human learning. December 18, 1968. (Journal of Verbal Learning and Verbal Behavior, 1969, 8, 507-511.)

140   R. C. Atkinson, J. E. Holmgren, and J. F. Juola. Processing time as influenced by the number of elements in the visual display. March 14, 1969. (Perception and Psychophysics, 1969, 6, 321-326.)

141   P. Suppes, E. F. Loftus, and M. Jerman. Problem-solving on a computer-based teletype. March 25, 1969. (Educational Studies in Mathematics, 1969, 2, 1-15.)

142   P. Suppes and M. Morningstar. Evaluation of three computer-assisted instruction programs. May 2, 1969. (Computer-assisted instruction. Science, 1969, 166, 343-350.)

143   P. Suppes. On the problems of using mathematics in the development of the social sciences. May 12, 1969. (In Mathematics in the social sciences in Australia. Canberra: Australian Government Publishing Service, 1972. Pp. 3-15.)

144   Z. Domotor. Probabilistic relational structures and their applications. May 14, 1969.

145   R. C. Atkinson and T. D. Wickens. Human memory and the concept of reinforcement. May 20, 1969. (In R. Glazer (Ed.), The nature of reinforcement. New York: Academic Press, 1971. Pp. 66-120.)

146   R. J. Titiev. Some model-theoretic results in measurement theory. May 22, 1969. (Measurement structures in classes that are not universally axiomatizable. Journal of Mathematical Psychology, 1972, 9, 200-205.)

147   P. Suppes. Measurement: Problems of theory and application. June 12, 1969. (In Mathematics in the social sciences in Australia. Canberra: Australian Government Publishing Service, 1972. Pp. 613-622.)

148   P. Suppes and C. Ihrke. Accelerated program in elementary-school mathematics--The fourth year. August 7, 1969. (Psychology in the Schools, 1970, 7, 111-126.)

149   D. Rundus and R. C. Atkinson. Rehearsal processes in free recall: A procedure for direct observation. August 12, 1969. (Journal of Verbal Learning and Verbal Behavior, 1970, 9, 99-105.)

150   P. Suppes and S. Feldman. Young children's comprehension of logical connectives. October 15, 1969. (Journal of Experimental Child Psychology, 1971, 12, 304-317.)

151   J. H. Laubsch. An adaptive teaching system for optimal item allocation. November 14, 1969.

152   R. L. Klatzky and R. C. Atkinson. Memory scans based on alternative test stimulus representations. November 25, 1969. (Perception and Psychophysics, 1970, 8, 113-117.)

153   J. E. Holmgren. Response latency as an indicant of information processing in visual search tasks. March 16, 1970.

154   P. Suppes. Probabilistic grammars for natural languages. May 15, 1970. (Synthese, 1970, 11, 111-222.)

155   E. M. Gammon. A syntactical analysis of some first-grade readers. June 22, 1970.

156   K. N. Wexler. An automaton analysis of the learning of a miniature system of Japanese. July 24, 1970.

157   R. C. Atkinson and J. A. Paulson. An approach to the psychology of instruction. August 14, 1970. (Psychological Bulletin, 1972, 78, 49-61.)

158   R. C. Atkinson, J. D. Fletcher, H. C. Chetin, and C. M. Stauffer. Instruction in initial reading under computer control: The Stanford project. August 13, 1970. (In A. Romano and S. Rossi (Eds.), Computers in education. Bari, Italy: Adriatica Editrice, 1971. Pp. 69-99. Republished: Educational Technology Publications, Number 20 in a series, Englewood Cliffs, N. J.)

159   D. J. Rundus. An analysis of rehearsal processes in free recall. August 21, 1970. (Analysis of rehearsal processes in free recall. Journal of Experimental Psychology, 1971, 89, 63-77.)

160   R. L. Klatzky, J. F. Juola, and R. C. Atkinson. Test stimulus representation and experimental context effects in memory scanning. (Journal of Experimental Psychology, 1971, 87, 281-288.)

161   W. A. Rottmayer. A formal theory of perception. November 13, 1970.

162   E. J. F. Loftus. An analysis of the structural variables that determine problem-solving difficulty on a computer-based teletype. December 18, 1970.

163   J. A. Van Campen. Towards the automatic generation of programmed foreign-language instructional materials. January 11, 1971.

164   J. Friend and R. C. Atkinson. Computer-assisted instruction in programming: AID. January 25, 1971.

A Rationale and Description of the BASIC Instructional Program

by

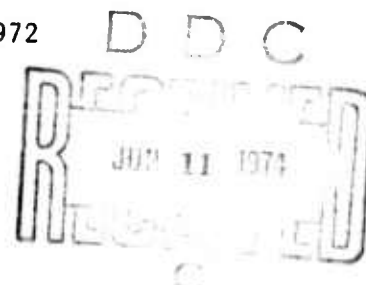Avron Barr, Marian Beard, and Richard C. Atkinson

Summary

A BASIC Instructional Program is being developed as a vehicle for research in tutorial modes of computer-assisted instruction (CAI). Several design features will be appropriate to training in other technical areas and applicable in other instructional settings where the development of analytic and problem-solving skills is a goal.

Methods are incorporated for monitoring and aiding the student as he works on programming problems in the BASIC language. The instructional program developed can be used to investigate schemes for optimizing problem presentation and giving assistance during problem solving based on a model of the student's abilities and difficulties. Previous experience in the instructional and technical aspects of teaching a programming language indicates that a course in computer programming can be designed to help the student acquire programming concepts in a personalized and efficient manner as he develops skills at increasingly advanced levels.

---

1

A major goal of the research project is to increase the sophistication with which the instructional program monitors the student's work and responds to it with appropriate hints and prompts. One aspect of such work is the utilization of algorithms for checking the correctness of a student procedure. Limited but sufficient program verification is possible through simulated execution of the program on test data stored with each problem. Within the controllable context of instruction, where the problems to be solved are predetermined and their solutions known, simulated execution of the student's program can effectively determine its closeness to a stored model solution.

The BASIC Instructional Program (BIP) is written in SAIL (VanLehn, 1973), a versatile, ALGOL-like language, implemented exclusively at present on the DEC PDP-10. SAIL includes a flexible associative sublanguage called LEAP (Feldman, Low, Swinehart, & Taylor, 1972), which was used extensively to build BIP's information network. The course is now running on the PDP-10 TENEX timesharing system at IMSSS and is presently being offered as an introductory programming course at DeAnza College in Cupertino and the University of San Francisco. The collected data are being used to modify the problems and the "help" sequences in preparation for a more controlled experimental situation planned for the next academic year.

## Overview of IMSSS Research in Tutorial CAI

The Institute has been involved in CAI projects in computer programming and in tutorial CAI in other technical areas since 1968. Work in teaching computer programming began with the development of a high-school-level CAI course in machine language programming (Lorton & Slimick, 1969). The project, called SIMPER, taught programming via a simulated three-register machine with a variable instruction set. Later, lessons in the syntax of the BASIC language were added to the curriculum. Programming problems using BASIC were presented, but the student solved them by linking to a commercial BASIC interpreter, without receiving assistance or analysis of his efforts from the instructional program.

In 1970 the Institute developed a much larger CAI curriculum for a new course to teach the AID programming language at the introductory undergraduate level. This course has been used in colleges and junior colleges as a successful introduction to computer programming (Friend, 1973; Beard, Lorton, Searle, & Atkinson, 1973). However, it is a linear, "frame-oriented" CAI program and cannot provide individualized instruction during the problem-solving activity itself. After working through lesson segments on such topics as syntax and expressions, the student is assigned a problem to solve in AID. He must then leave the instructional program, call up a separate AID interpreter, perform the required programming task, and return to the instructional program

3

with an answer. As he develops his program directly with AID, his only source of assistance is the minimally informative error messages provided by the interpreter.

In recent years, developments in interactive CAI and in artificial intelligence have enabled teaching programs to deal more effectively with the subject matter they purport to teach, in effect, to "know" their subject better. The generative CAI programs developed by Carbonell and others (Carbonell, 1970; Collins, Carbonell, & Warnock, 1973) employ a semantic network interrelating a large factual data base. Instruction then takes the form of a dialogue in which the program can both a) construct, present, and evaluate the answers to a multitude of questions, and b) answer questions posed by the student. An interesting generative CAI program in digital logic and machine-language programming has been developed by Elliot Koffman at the University of Connecticut (Koffman & Blount, 1973). Another course in programming is being written by Jurg Nievergelt for the PLATO IV system at the University of Illinois (Nievergelt, Reingold, & Wilcox, 1973).

Two CAI courses developed at IMSSS are capable of dealing in a sophisticated way, both with their subject matter and with the student. These courses provide instructive interaction throughout the problem-solving activity by performing operations specified by the student, evaluating the effect of the operations, and, on request, suggesting a next step in the solution.

4

The first of these, a CAI program for teaching elementary mathematical logic, is described in a report by Adele Goldberg (1973). An experimental version of the program employed a heuristic theorem-prover as a proof-analyzer to generate appropriate dialogue with students who needed help with a proof. "The proof-analyzer mocks the adaptive behavior of a human tutor; it can determine relevant hints when a student requires help in completing a solution, and it can encourage the student to discover diverse solution paths." While the prover was limited, the heuristics it supplied were more natural than those that might be supplied by more powerful, resolution-based theorem-provers. A version of this program without a theorem-prover has been used successfully as a primary source of instruction in an introductory symbolic logic course at Stanford for the past three years.

A CAI course described in Kimball (1973) uses symbolic integration routines and an algebraic expression simplifier to assist students in learning introductory integration techniques. The program stresses development of student heuristics by performing most of the tedious computations (substitutions, integration by parts, etc.) for the student after he has completely specified the parameters. An attempt is made to estimate each student's knowledge of integration methods individually, in order to select problems dynamically.

5

## The BIP Course

The goal of a tutorial CAI program is to provide assistance as the student attempts to solve a problem. The program must contain a representation of the subject matter that is complex enough to allow the program to generate appropriate assistance at any stage of the student's solution attempt. Both the logic and the calculus courses approach this goal. However, computer programming is an activity fraught with human variability, and how an individual calls on his programming skills to write a program is not so clear as, for example, how he uses logic in achieving a proof. Furthermore, the difficulty of describing and verifying program segments precludes the kinds of solution analysis performed by the logic and calculus courses. BIP contains a representation of information appropriate to the teaching of computer programming that allows the program to provide help to the student and to perform a limited, but adequate analysis of the correctness of his program as a solution to the given problem. As a vehicle for research in instructional strategies, BIP will serve as both a teaching and a learning tool.

To the student seated at his terminal, BIP looks very much like a typical timesharing BASIC operating system. The BASIC interpreter, written especially for BIP, analyzes each program line after the student types it and notifies the student of syntax errors. When the student runs his program, it is checked for structural illegalities,

and then, during runtime, execution errors are indicated. A file storage system, a calculator, and utility commands, like TIME, are available.

Residing above the simulated operating system is the "tutor," or instructional program. It overlooks the entire student/BIP dialogue and motivates the instructional interaction. In addition to selecting and presenting programming tasks to the student, the instructional program identifies the student's problem areas, suggests simpler subtasks, gives hints or model solutions when necessary, offers debugging aids and a facility for communicating with the Stanford staff, and supplies incidental instruction in the form of messages, interactive lessons, or, most often, manual references. Each student receives a BIP manual that introduces him to programming, the BIP system, and the syntax of BIP's version of BASIC. The manual serves as the student's primary source of information throughout the course.

At BIP's core is an information network that embodies the interrelations of the concepts, skills, problems, subproblems, pre-requisites, BASIC commands, remedial lessons, hints, and manual references. We believe that with a sufficient student history, the network can be successfully applied to a student learning model to present an individualized problem sequence, to control the frequency and type of assistance given during programming, and to identify problem areas. Our experimental work will compare different student models and decision algorithms, including a "free" or "student-choice"

mode where the student is given enough information for him to select his own problems.

Figure 1 illustrates schematically the interacti ns of the parts of the BIP program. Each of these is discussed in detail below.

# BIP: Information Flow Diagram



Figure 1

9
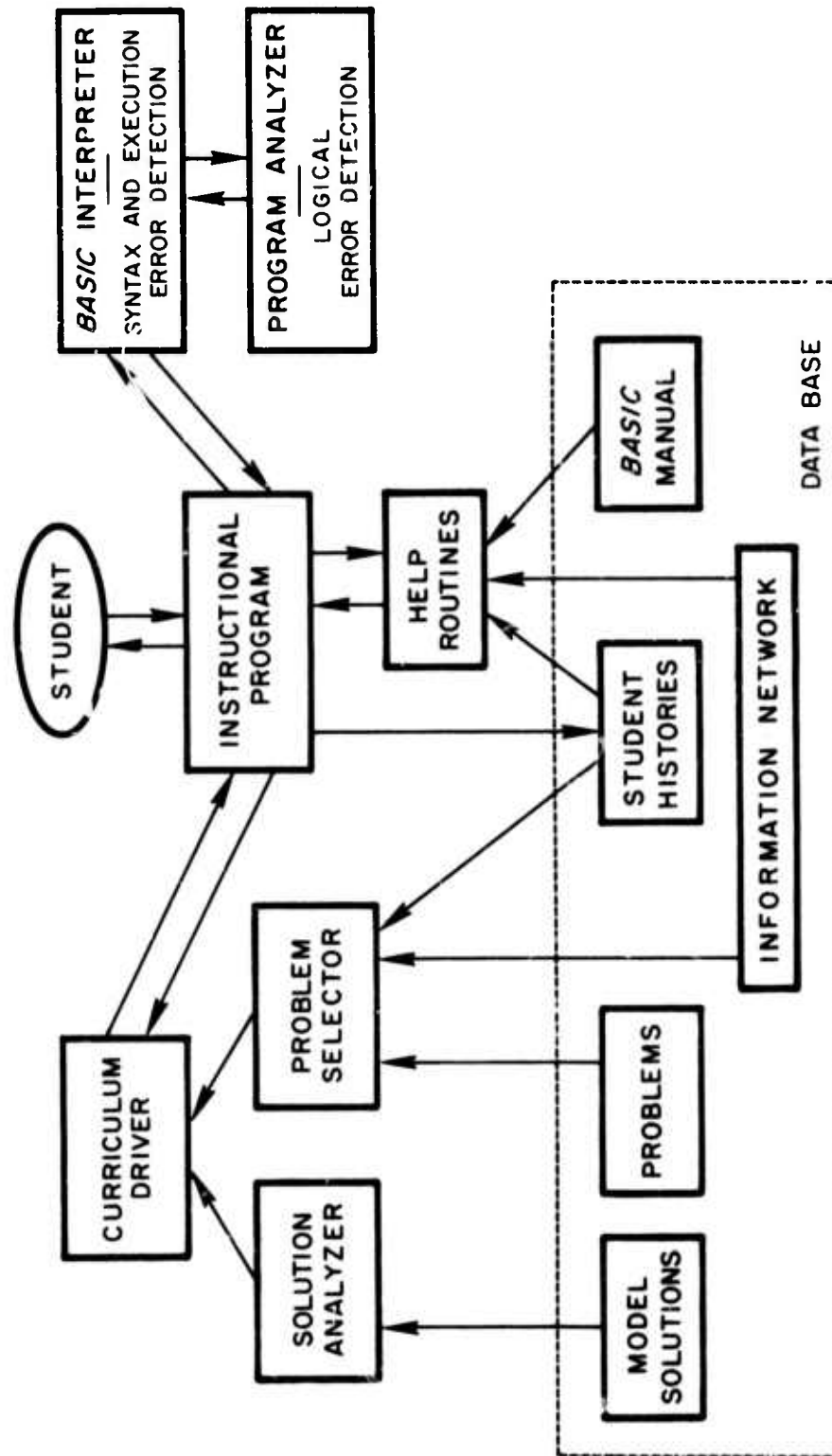
## The BASIC Interpreter, Error Detection, Assistance, Debugging Aids

BIP's interpreter was specially designed to allow the instructional program full access to the student's programs and his errors. It handles a complete subset of BASIC. During a student's work on a task, each of the BASIC operators can be temporarily deactivated as required for pedagogical purposes. For example, during a simple task whose instructions require the use of a FOR...NEXT loop and in which no other branching is necessary, IF statements will not be accepted. The student is reminded that he is to use FOR...NEXT to form his loop.

Immediately after the student enters a line, syntax analysis is performed. (Any student entry beginning with a number is assumed to be a line of BASIC code.) If a syntax error is discovered, an error message ("illegal print list," "missing argument for INT") is sent to the student, the error number is retained by the instructional program for reference if the student requests more help, and the line is rejected.

If he does not understand the syntax mistake immediately, the student can request one of three types of assistance by beginning his next line with a question mark:

?         An explanatory message stored for this syntax error is
          printed. Repeated requests summon different messages
          until they are exhausted.

?REF    A manual reference covering the particular syntax involved in the error is printed for the student.

?LES    An interactive lesson, relevant to the syntax error, is presented. The lesson provides drill-and-practice instruction on the student's syntactic difficulty.

Once the student has entered a syntactically legal program, he can have it executed in one of three formats, Two of which involve debugging aids. After his request, and before the actual execution, the student's program is checked for illegal program structure (e.g., a missing END statement, or illegally nested loops) by a routine we call ERR DOKTOR. If all is well, one of the three modes of program execution is initiated:

RUN    The student's program is executed, as in standard BASIC implementations, in the order of its line numbers.

TRACE    (A debugging option) The student controls execution of the program using the standard interactive debugging technique of stepping through it one line at a time. As a line is executed, its number is printed. This allows direct observation of the execution sequence of such structures as loops and conditional branches.

11

When an assignment statement, which initializes or changes the value of a variable, is executed, the variable and its new value are printed with the line number. The student can easily see the "internal" activity of the program, which would otherwise be visible to him only by means of extra statements printing interim results.

By specifying inclusive line numbers, the student can TRACE a selected section of his program. This is useful when he is satisfied with other parts of the program and wishes to avoid the time-consuming process of tracing those parts.

When CRT display units are used as the student terminals in place of teletypes, the format is slightly different. The program listing appears on one half of the screen, with the currently executed line blinking. The variables and their values will appear on the other half of the screen as assignment statements are executed.

FLOW    (The second debugging aid) This option is available on CRT display terminals. FLOW differs from TRACE in that a flowchart representation of the program appears in place of the program listing. As the student steps

through the execution, the element of the flowchart representing the current line blinks. Variables and their values appear in the other half-screen. The FLOW option involves the interface of a flowchart generating routing (under development) with the tracing procedure.

There are four ways in which any mode of execution can terminate. Normal termination follows execution of a BASIC END or STOP statement. The student is told that "execution terminated at line xxx." Alternatively, the student can abort execution by typing a control key; BIP responds with the message "execution aborted at line xxx." The third cause of termination is excessively long running, which is at present determined on the basis of the count of the number of lines executed. A message indicating BIP's suspicion of an infinite loop is printed.

Finally, runtime errors terminate execution. If an unassigned variable, illegal GOTO, or other error is discovered, an appropriate error message is printed, the error number is stored by the IP, and execution terminates. The student may then request the same three types of assistance for execution errors discussed under syntax errors above.

## Goals of the Curriculum

Prior experience with CAI in programming at the college level has convinced us that many students who wish to learn the fundamental principles and techniques of programming have limited mathematical backgrounds. More important, their confidence in their own abilities to confront problems involving numeric manipulation is low. The scope of the BIP curriculum, therefore, is restricted to teaching the most fundamental of programming skills and does not extend to material requiring mathematical sophistication.

The curriculum is designed to give the student practice and instruction in developing interactive programs in order to expose him to uses of the computer with which he may well be unfamiliar. BIP guides the student in construction of programs that he can "show off." The emphasis is on programs that are engaging and entertaining, and that can be used by other people. As the student writes his programs, he keeps in mind a hypothetical user, a person who will use the student's program for his own purposes and to whom the performance of the program must be intelligible. The additional demands for clarity and organization forced by interactive programming, as well as the increased noticeability of bugs are valuable, as are the added motivational effects.

Numerous texts were examined as possible sources for the necessary programming principles to be developed in an introductory

14

course and for the problems that illustrate those principles. We incorporated ideas from general computer science textbooks (Forsythe, Keenan, Organick, & Sternberg, 1969), from the excellent notes for an introductory programming course that were oriented toward the ALGOL language but whose examples were easily generalized (Floyd, 1971), and from books and notes dealing specifically with BASIC (Albrecht, Finkel, & Brown, 1973; Coan, 1970; Kemeny & Kurtz, 1971; Nolan, 1969; Wiener, 1972; various publications of the People's Computer Company). In addition, problem sets from Stanford University's introductory computer science courses were collected and examined.

In general, the curriculum provides useful, entertaining, and practical computer experience for students who are not necessarily mathematically oriented. It gives them the opportunity to develop programming skills while working on problems that are challenging but not intimidating, in which the difficulties stem from the demands of logical program organization rather than from the complexities of the prerequisite mathematics. The curriculum text is listed in Appendix A.


## The Curriculum Driver


The curriculum is organized as a set of discrete programming problems called tasks, whose text includes only the description of the problem, not lengthy descriptions of programming structures or

15

explanations of syntax. There is no default ordering of the tasks; they are not numbered. The decisions involving a move from one task to another can be made only on the basis of the information about the tasks (skills involved, prerequisites required, subtasks available) stored in BIP's information network.

A student progresses through the curriculum by writing and running a program that solves the problem presented on his terminal. Virtually no limitations are imposed on the amount of time he spends, the number of lines he writes in his program, the number of errors he is allowed to make, the number of times he chooses to execute the program, or the changes he makes within it. The task he is performing is stored on a stacklike structure, so that he may work on another task and return to the previous task automatically. All BIP commands (listed in Appendix B) are available to the student at all times. The following commands deal specifically with the curriculum driver:

HINT    When a student experiences difficulty with a task,
        several levels of help are available. HINT retrieves
        problem-specific hints from a set stored in the
        network.

SUB     If, after pondering the available hints, a method of
        attack has still not occurred to the student, he can
        have the task broken into conceptually simpler

16

subtasks. These are presented one at a time as tasks, while the main task is pushed onto the stack structure. When the student completes a subtask, BIP returns him automatically and explicitly to the larger problem.

ENOUGH   If he understands the demands of the larger program during his work on the subtask, he can type ENOUGH and return to the larger task from which he started. Outside of a subtask, typing ENOUGH terminates work on the current task without giving the student credit for having completed it.

MODEL   After exhausting all hints and subtasks available for a given task, the student can request that BIP suggest a model solution. The model stored for each task is intended to be easily understood, and correct, but it is not necessarily the shortest or most elegant solution.

RESET   Typing RESET clears the task stack of all the tasks on which he has been working, so the student can start fresh if he wants.

MORE   When he feels that he has solved the problem, the student types MORE and BIP takes over, as described in the "Solution Analysis" Section.

17

The curriculum structure allows for a wide variety of student aptitudes and skills. Most of the curriculum-related options are designed with the less competent, less confident student in mind. A more independent student may simply ignore the options. Thus BIP gives all students the opportunity to determine their own individual challenge levels simply by making assistance available, but not inevitable.

BIP offers the student considerable flexibility in making task-related decisions. As explained above, he may ask for hints and subtasks to get started in solving the given problem, or he may ponder the problem on his own, using only the manual for additional information. He may request a different task by name, in the event that he wishes to work on it immediately, either completing the new task or not, as he chooses. On his return, BIP tells him the name of the again current task and allows him to have its text printed to remind him of the problem he is to solve. The student may request the model solution for any task at any time, but BIP will not print the model for the current task, unless he has exhausted the available hints and subtasks. Taken together, the curriculum options allow for a range of student preferences and behaviors; this flexibility will be put to use in the experiments referred to earlier, comparing student-selected and BIP-determined curriculum decisions.

## Solution Analysis

At present a student is not considered to have completed a problem if he has not executed his current program successfully. BIP "knows" at all times (a) whether an executable, syntactically legal program exists, (b) whether the student has executed that program, (c) whether execution errors have occurred, and (d) whether the student has made changes or additions since the last execution. The student's history will be updated to indicate successful completion of a task only if he has succeeded in an error-free execution of the most recent version of his program.

Error-free execution of a program is no guarantee that the program correctly solves the problem presented. Program analysis is an embryonic art, and BIP is not capable of "understanding" a student's programs in the fullest sense implied by current research in artificial intelligence. We are, however, investigating two promising potential additions to BIP that are expected to provide sufficient solution analysis for pedagogical purposes, without involving a full-scale application of program verification techniques. The results of the two analysis efforts should allow BIP to give the student an indication of (a) the kinds of test values that his program fails to handle properly, and (b) the kinds of programming structures that his program should have but doesen't.

The first analysis scheme we will apply is simulated execution of

19

the student's program on test data, comparing its output with that of one or more model solutions. A preliminary dialogue will establish the variable names that the student has used for critical input/output variables. Clearly this method will often fail to indicate all of the student's logical errors, but we are hopeful that in cases where known problems call for fairly simple solutions, an analysis will succeed in discovering particular kinds of problem-specific errors. The second method involves comparison of program flow diagrams, again matching the student's effort against a model solution. BIP generates this internal representation of the student's program to both check for legal program structure and draw flowcharts as a pedagogical/debugging tool, and we are investigating methods by which the schemas of different programs can be compared.

## BIP's Information Network

Task selection, remedial assistance, and problem area determination, BIP's "tutorial" activities, require that the program have a flexible information store interrelating the tasks, hints, manual references, etc. This store has been built using the associative language LEAP (Feldman, 1972). The network is constructed using an ordered-triple data structure and is best described in terms of the various types of nodes:

20

TASKS    All curriculum elements exist as task nodes in the network. They can be linked to each other as subtasks, prerequisite tasks, or "must follow" tasks.

SKILLS    The skill nodes are intermediaries between the concept nodes and the task nodes (see Fig. 2). Skills are very specific, e.g. "concatenating string variables" or "incrementing a counter variable." By evaluating access on the individual skills, the program estimates competence levels in the concept areas. In the network, skills are related to the tasks that require them and to the concepts that embody them.

CONCEPTS

The concept areas covered by BIP are, for the time being, the following:

> Interactive programs
> Variables and literals (numeric and string)
> Expressions (algebraic, string, and Boolean)
> Input and output
> Program control – branching
> Repetition – loops
> Debugging
> Subroutines
> Arrays (one dimensional)

The specific implementation of concept nodes in the network is not completely determined, but the links will be to the skills and only through them to the tasks.

21

BASIC OPERATORS

Each BASIC operation (PRINT, LET, ...) is a node in the network. The operations are linked to the tasks in two ways: first as elements that must be used in the solution of the problem, and second as those that must not be used in the solution. (These are temporarily disabled in the interpreter.)

HINTS   The hint nodes are linked to the tasks they may be helpful in. Each time a new skill, concept, or BASIC operator is introduced, there is an extra hint that gives a suitable manual reference.

ERRORS   All discoverable syntax, structural, and execution errors exist as nodes in the network, and are linked to the relevant help messages, manual references and remedial lessons.

## A Segment of BIP's Information Network



PROBLEMS

WRITE A PROGRAM THAT : PRINTS THE NAME OF A VARIABLE AND ITS VALUE

WRITE A PROGRAM THAT : PRINTS THE VALUE OF A VARIABLE

WRITE A PROGRAM THAT : PRINTS THE WORD "CAT"

SKILLS

ASSIGNING LITERALS

PRINTING VARIABLES

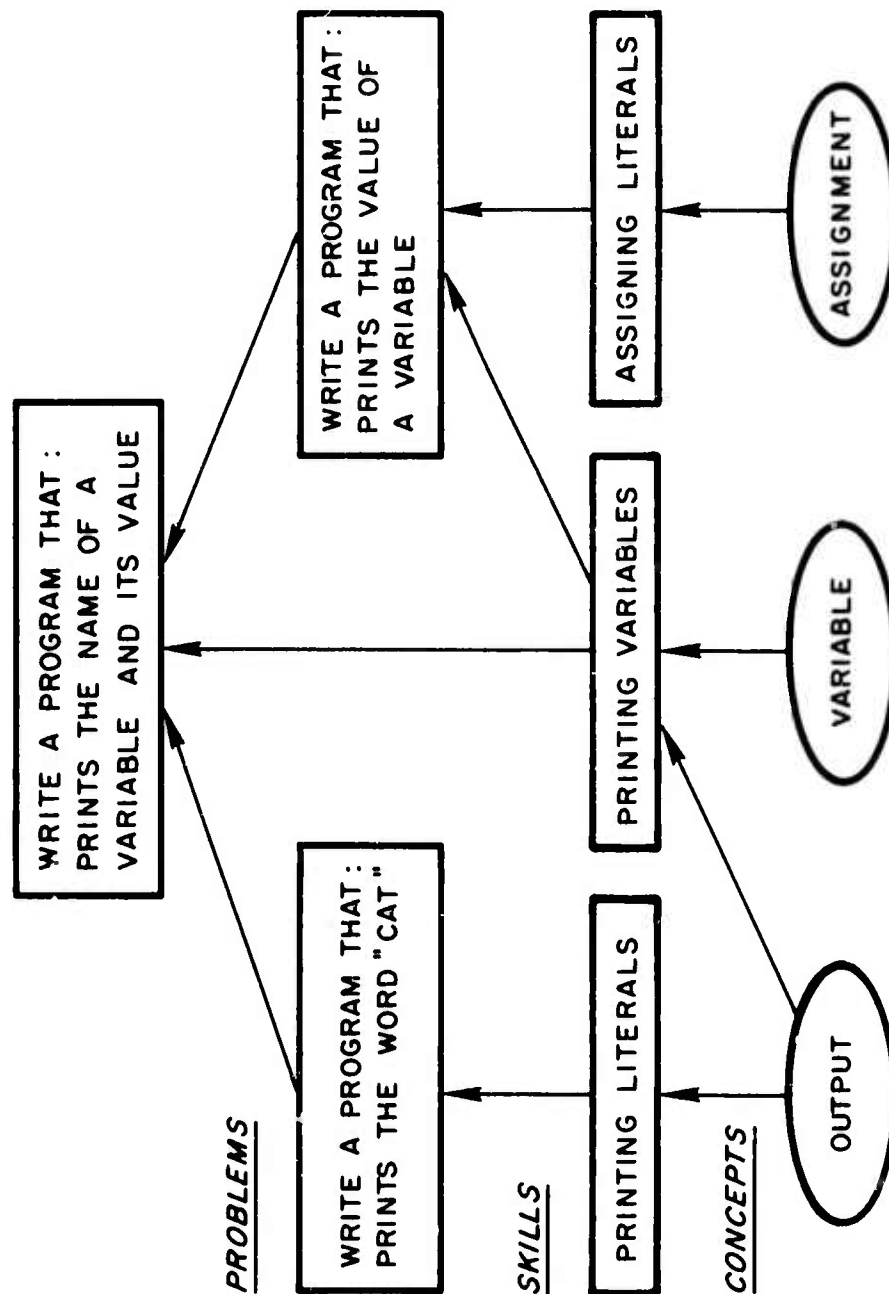PRINTING LITERALS

CONCEPTS

ASSIGNMENT

VARIABLE

OUTPUT

Figure 2

23

Upon completion of a task, the student is given a posttask interview in which BIP presents the model solution stored for that problem. (The student is encouraged to regard the model as only one of many possible solutions.) BIP asks the student whether he has solved the problem, then asks, for each of the skills associated with the task, whether he needs more practice involving that skill. The responses are stored and used in future BIP-generated curriculum decisions. BIP then informs the student that he has completed the task, and either allows him to select his next task by name (from an off-line printed list of names and problem texts), or selects it for him.

An example of the role of the Information Network in BIP's tutorial capabilities is the BIP-generated curriculum decisions mentioned above. By storing the student's evaluation of his own skills, and by comparing his solution attempts to the stored models, BIP can be said to "learn" about each student as a individual who has attained a certain level of competence in the skills associated with each task. BIP can then search the network to locate the skills that are appropriate to each student's different abilities and to present task that incorporate those skills. The network provides the base from which BIP can generate decisions that take into account both the subject matter and the student, behaving somewhat like a human tutor in presenting material that either corrects specific weaknesses or challenges and extends particular strengths, proceeding into as yet unencountered areas.

24

## The BIP Manual

It is tedious and probably ineffective to present voluminous description, explanation, and examples from the computer directly on the terminal. We have chosen instead to present this material to the student in a printed manual of approximately 50 pages. The manual includes complete instructions on the operation of the course (signing on, dealing with the terminal, dealing with BIP), a general introduction to computers (their capabilities and the concepts involved in programming languages), and the syntax of BIP's BASIC, complete with examples and suggestions for the appropriate uses of each of the BASIC statements.

All programming terms used in the manual and in the tasks are defined briefly in the glossary at the end of the manual. References to the relevant sections of the manual are included in each glossary entry. All words that have precise programming meanings different from their normal English meanings are listed.

We believe that when the student encounters another programming language with which he is not familiar his primary resource will be the manual for that language. He is not likely to have an instructor or a CAI course at hand, and the principal means by which he will learn the new language will be through his own experimentation, guided by the explanations and examples in the manual. Experience with BIP (with its frequent cross-references to the manual) will, we hope, give

the student a degree of confidence and ease in finding his way in other situations, when the manual may be his only guide.

## Miscellaneous Options Available to the Student

Several additional features are available to BIP students:

CALC    All BASIC expressions (numeric, string, and Boolean) can be evaluated by this BIP command. This is not only a convenience, freeing the student from having to write and run a complete program to make a simple calculation, but it is also useful as a debugging aid.

FILE SYSTEM: FILES, SAVE, GET, MERGE, KILL

BIP allows each student to save permanently as many as four programs, with names he designates. This gives him the opportunity to work on an extended programming project and simultaneously to accumulate his work from each session at the terminal. He can obtain a listing of his file names, with their most recent write dates, and his saved programs are always immediately retrievable for modifications or additions.

FIX     This feature allows the  student to send a  message to

the programmers at Stanford.  It gives him a chance to

communicate difficulties and confusions and helps both

to improve BIP's interaction abilities and to identify

and locate errors in the program.  The  convenience of

typing  a message  or  complaint while  seated  at the

terminal  encourages  students  to  provide   us  with

immediate and valuable feedback.

LOG-IN MESSAGE

Although not strictly  a student option,  this feature

prints a stored message to each student as he signs on

to the course.  The message is updated  frequently and

gives  information  about  revisions  to  the  course,

responses to messages left by students, and notices of

meetings with Stanford personnel at which students may

discuss  questions  too  complex  to  handle  in short

written messages.


We are  in continuous communication  with students who  are using

the course and whose suggestions regarding more flexible, intelligible

interaction  with  BIP  have  generaged  several  improvements.   Past

experience  has shown  that superficial  problems in  dealing  with an

instructional program can become significant barriers to acquiring the

27

concepts and skills presented by the program, and we continue to make

additions to BIP to eliminate frustrating confrontations between the

student and the uncomprehending machine.

## APPENDIX A

## THE BIP CURRICULUM

The following is the text  for all tasks, hints, and  subtasks in the pilot-year curriculum.  Some explanatory remarks are in order.

(1) The tasks appear in the order in which BIP would present them if it had no access to the student history.  This order is modified in two ways: either by the  student's choice of a particular task,  or by BIP's decision based on the student's previous work.

(2) A MORT is a continuation of the original problem, calling for a modification  or extension  of the  program just  completed.  Within this listing, the text of each task is followed by the hints  and sub- tasks associated  with it;  the MORTs  of the  task are  printed next, followed by their own hints and subtasks.

(3)  Because some  tasks require  similar skills  and strategies, some hints and subtasks are  associated with more than one  main task, and thus they appear more than once in this listing.

(4) References to Section XXX refer to the BIP manual supplied to each student.

(5) Terms enclosed in  asterisks (e.g., *print* )  call attention to  the special  use of  that  term.  All  such terms  are  listed and explained in the glossary of the manual.

TASK PR1:

Before you start the first problem, be sure to read about the BIP course in the BIP manual.

Then read about the structure of BASIC programs.

Type "MORE" when you're ready.

MORT:

Now write a *program* to *print* the *number* 6 on your teletype. Then *run* the *program*.

TASK OP1:

SCRATCH your old program. Then write and *run* a *program* that *prints* the *sum* of 6 and 4.

MORT:

Now modify the program to do each of the following:
print the *difference*
print the *product*
print the *quotient*

HINT:

'Sum' means addition
'Difference' means subtraction
'Product' means multiplication
'Quotient' means division

TASK VN1:

SCRATCH your old program. then write a program that:
1. *Assigns* the *value* 6 to a *numeric variable* N.
2. *Prints* the value of this variable.

TASK VX1:

Write a program that:
1. Assigns the value 6 to N.
2. Prints the sum of N and 4.

TASK VX2:

Write a program that:
1. Assigns the value 6 to M.
2. Assigns the value 4 to N.
3. Prints the sum, difference, product and quotient of M and N.

HINT:
'Sum' means addition
'Difference' means subtraction
'Product' means multiplication
'Quotient' means division


TASK IN1:

Write a program that:
1. Allows the user to *input* a value to M and a value to N.
2. Prints their sum, difference, product and quotient.


TASK IN2:

Write a program that:
1. Allows the user to choose the arithmetic operation he wants the program to perform.  He should type 1 to add, 2 to subtract, 3 to multiply, or 4 to divide.  Use the variable X for this code number.
2. Allows him then to input the values for M and N.
3. Prints out the result of the operation he asked for when he gave a value to X.  For example, if he typed 4, you should print the quotient of the numbers he gave for M and N.

*SAVE* this program  when you get  it to work.   It will help you later.

HINT:
Read about **IF . . THEN** statements in Section III.11.

HINT:
Depending on the value  of X, the program should  do one of four things. Get X first, then get M and N.  then use X to decide which **PRINT** statement to *branch* to.

SUB:

You need a program that can make decisions, then you can incorporate the arithmetic operations into it. Translate the following into BASIC (it is definitely not BASIC now), and run it:
  1. let the user type a number between 1 and 4.
  2. if the number is 1, jump to 7
  3. if the number is 2, jump to 9
  4. if the number is 3, jump to 11
  5. the number must be 4, so print "YOU TYPED A 4!"
  6. jump to the end of the program
  7. the number is 1, so print "YOU TYPED A 1!"
  8. jump to the end
  9. print "YOU TYPED A 2!"
  10. jump to the end
  11. print "YOU TYPED A 3!"
  12. the end
Once this program works, type "MORE" and return to the main task.


MORT:

Now fix up the program so that it prints out questions and little messages that tell the user:
a) What to do (e.g. "TYPE 1 FOR ADDITION",...).
b) What the result represents (e.g. "THE SUM IS ...").

HINT:

Type MODEL IN2 and copy what you need, then make the necessary additions to it.


MORT:

Modify the program once again so that it keeps *looping* back to the beginning until the user inputs a 0 for the operation code.

HINT:

Type MODEL IN2 and copy what you need, then make the necessary additions to it.

HINT:

You need two more statements:
an **IF . . THEN** after the "INPUT X" that jumps to the end if X is zero,
a **GOTO** back to the line with the instructions.

TASK ST1:

Please read about *strings* before you get confused.
Write (and run) a program that prints the string
"SCHOOL".

TASK VS1:

Assign the value "HORSE" to the *string variable* X$ and
print the value of X$.

TASK SX1:

Allow the user to **INPUT** the value of the string
variable X$. then print that value. (Your program will
just "echo" what the user types, whether he types a
number or a word.)

MORT:
Read about *concatenation* of strings.
*Concatenate* the word "OKAY" (or any word you like) to
the user's input.  Print the result.

TASK SX2:

Assign the string "DOG" to X$ and the string "HOUSE" to
Y$.  Print the *concatenation* of X$ and Y$.

HINT:
Concatenation is in Section III.6.  Type the & character
with The shift key and the 6 key.

MORT:
(Keep the same string values of X$ and Y$.)
Assign the *concatenation* of Y$ and X$ to the variable
Z$.  Print the value of Z$.

MORT:
(Still with the same values of X$ and Y$.)
"HOUSEDOG" should have a space between the words.
*Concatenate* a space between Y$ and X$ and print the
result.

33

HINT:
>    The literal "A" prints the letter A
>    What character between quotes will print as a space?

## TASK SX3:

>    Allow the user to input the values of X$ and Y$.
>    Concatenate the strings with a space between them and
>    print the result.

## TASK SX4:

>    Let the user make up a sentence.
>    1. Ask him how many words he wants to have in the
>    sentence.
>    2. Let him input those words, one at a time.
>    3. After each input, concatenate a space and his latest
>    word into a string variable. Use X$ for the input word,
>    and use S$ to hold all the concatenations.
>    4. After you have looped around the specified number of
>    times, print his sentence.

HINT:
>    make S$ equal to the string version of nothing, like
>    this: S$ = "" outside the loop.
>    Inside the loop, use S$ to accumulate the sentence: S$ =
>    S$ & " " & X$

SUB:
>    A very important sub task:
>    Write a program with a little loop. The "work" of the
>    loop is just to print the value of the loop's index.
>    When you run the program, it should look like it is
>    counting from 1 to the top value. Use whatever top
>    value you like.

SUB:
>    Very important:
>    Write a loop that prints the value of its index. Start
>    the loop at 1, but let the user give the top value. You
>    can add to this program, making the loop do some real
>    work, and the work will then be done as many times as
>    the user likes.

TASK INT1:

Rewrite your calculator so that the user can type
"+" for addition
"-" for subtraction
"*" for multiplication
"/" for division
to tell the calculator which operation to perform. You may have *SAVED* your calculator program; if so, use
GET
to retrieve it.

HINT:
Type MODEL IN2 and copy what you need, then make the necessary additions to it.

SUB:
You need a program that can make decisions about strings, then you can incorporate the arithmetic operations into it. Write a program that asks the user to type any character. If he typed a ! mark, the program should say "YOU TYPED A !" . If he typed something else, it should say "YOU DID NOT TYPE A !"


TASK XMAS:

On the first day of Christmas, someone's true love sent him/her a partridge in a pear tree (one gift). On the second day, the true love sent two turtle doves in addition to another partridge (three gifts on the second day). This continued through the 12th day, when the true love sent 12 lords, 11 ladies, 10 drummers, ..... all the way to yet another partridge. Write a program that computes and prints the total number of gifts sent on that 12th day.

HINT:
This program requires a loop. Each execution of the loop involves accumulating the value of the index into a total.

HINT:
Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.
In words, total equals total plus another value.

35

SUB:

A very important sub task:
Write a program with a little loop. The "work" of the loop is just to print the value of the loop's index. When you run the program, it should look like it is counting from 1 to the top value. Use whatever top value you like.


MORT:

Modify your program so that it prints the total gifts for each day. (Day 1 = 1 gift, Day 2 = 3 gifts, Day 3 = 6 gifts, etc.)

HINT:

You need one statement that prints the value of the index (the number of days) and the accumulated total of gifts.


MORT:

The user of your program has a true love who will send presents in the same way for as many days as the user wants. Let your user say how many days, and calculate the number of gifts sent on that day. (The generous true love may send presents for more than 12 days, if the user likes.)

SUB:

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.


TASK PAY:

A man is paid 1 cent the first day he works, 2 cents the second day, 4 cents the third, 8 cents the fourth, etc. (doubling his wage each new day). Calculate his wage for the 30th day.

HINT:

Say w is the variable for the wage. On the first day, W equals 1. For every day after that, W equals W * 2.

**MORT:**

Modify the program to calculate the total wages for the the month: sum of the first day plus the second day ..... plus the 30th day.

**HINT:**

You have a variable for each day's wage. You need another variable to accumulate the total.

**HINT:**

Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.
In words, total equals total plus another value.

**MORT:**

Your program's user has a contract with this man, for the same schedule of wages. Tell the user how much he will owe the man for any number of days he (the user) specifies.

**SUB:**

Very important:
Write a loop that prints the value of its index. Start the loop at 1, but let the user give the top value. You can add to this program, making the loop do some real work, and the work will then be done as many times as the user likes.

**TASK IT1:**

Write a program that counts (and prints) the number of odd numbers between 5 and 187 inclusive. For example, there are 3 odd numbers between 5 and 9 inclusive: they are 5, 7, and 9. And a program that counted those numbers would print something like this:

THERE ARE 3 ODD NUMBERS BETWEEN 5 AND 9

Do not print each odd number as you count it.

**HINT:**

Any odd number plus 2 equals the next odd number.

HINT:

You know the bottom and top values of the loop, but the point of the program is to see how many times the loop must be executed before it gets to the top. Use a counter inside the loop and add to it with each execution.

MORT:

Now find the sum of all those odd numbers you just counted.

HINT:

Finding a total or sum almost always means two things:
1. Setting a variable equal to zero outside a loop.
2. Accumulating into that variable within the loop.
In words, total equals total plus another value.

MORT:

Let the user specify a range, and tell him 1) how many odd numbers are in that range, and 2) the sum of those numbers. For example, you ask him for the lower limit (suppose he gives 9). Then you ask him for the upper limit (suppose he gives 17). The number of odd numbers in that range is 5 (9, 11, 13, 15, 17), and the sum is 65.

HINT:

The top and bottom values for the loop come from the user. The work of the loop is just to count how many times it is executed.

TASK IT2:

Find the number of integers greater than 99 and less than 278 that are divisible by 11. You don't need any division to do this.

HINT:

You know the bottom and top values of the loop, but the point of the program is to see how many times the loop must be executed before it gets to the top. Use a counter inside the loop and add to it with each execution.

MORT:
   Now find the sum of the numbers greater than 99 and less
   than 278 that are divisible by 11.

HINT:
   Finding a total or sum almost always means two things:
   1. Setting a variable equal to zero outside a loop.
   2. Accumulating into that variable within the loop.
   In words, total equals total plus another value.


TASK AV:

   Find the average  of 10 numbers.   Ask the user  to give
   the numbers, one at a time.

HINT:
   Finding a total or sum almost always means two things:
   1. Setting a variable equal to zero outside a loop.
   2. Accumulating into that variable within the loop.
   In words, total equals total plus another value.

HINT:
   The average of 10 numbers is their sum divided by 10.

SUB:
   A very important sub task:
   Write a program with  a little loop. The "work"  of the
   loop is  just to  print the value  of the  loop's index.
   When  you run  the program,  it should  look like  it is
   counting  from 1  to the  top value.   Use  whatever top
   value you like.


MORT:
   Modify  the program  to let  the user  specify  how many
   numbers he  wants to  average. Let  him type  that many
   numbers one at a time, then tell him the average.

HINT:
   The average of N numbers is their sum divided by N.

SUB:
   Very important:
   Write a loop that prints the value of its  index.  Start
   the loop at 1, but let the user give the top value.  You
   can add to  this program, making  the loop do  some real
   work, and the  work will then be  done as many  times as
   the user likes.


39

TASK GAS:

Write a program to calculate the user's gas mileage. He
recorded his car's mileage at the beginning of the trip,
and again at the end of the trip, when he bought some
amount of gas. Ask him for the starting and ending
mileages (and calculate the miles driven), then ask for
the number of gallons of gas he bought. Then tell him
his gas mileage (miles per gallon).

Example:  starting mileage = 5325
          ending mileage = 5550
          (miles driven = 5550 - 5325 = 225)
          gallons of gas = 9
          gas mileage = 225 miles / 9 gallons = 25 mpg.

MORT:

Each time the user buys gas, he records the mileage and
the gallons bought. Modify your program to ask him how
many times he bought gas; then ask for the mileage and
gallons he recorded each time. Accumulate the total
miles traveled and the total gallons, then print those
totals and the gas mileage. Test the program with some
very simple numbers to be sure that it calculates
correctly.

HINT:

You only need the starting mileage once. Total miles
equals the last mileage recorded minus starting mileage.
Keep a running total of gallons bought.

TASK GUESS:

Write a program that plays a guessing game. Generate a
random integer between 1 and 25 (read the manual first),
then let the user guess what the number is. Print
appropriate messages if his guess is too high or too
low, and give him another chance to guess. Congratulate
him for guessing correctly.

HINT:

Break this problem into parts. You need a loop whose
"work" is to get and compare the user's guess. Generate
the random number before the loop, and print the
correct-guess message after the loop.

SUB:

Forget about random numbers for now. Write a program that gets a number from the user and compares his number to 100. Print "HIGHER THAN 100!" or "LOWER THAN 100!" or "100 EXACTLY!" appropriately. Then you can put this part together with the other parts you need in the main task.

SUB:

Your program must get a number from the user again and again, until the input number equals some set value (the random number). For now, write a program that asks for a number and checks to see if that number equals 100. If it is 100, the program should stop; if not, it should ask for another input. Then you can fit this part into the main task.

MORT:

Add a feature to your program that tells the user how many guesses he needed. Three lines will do it: one to assign the value 0 to a counter variable, one to add to the counter each time he guesses, and one to print the value of the counter with some appropriate message.

MORT:

Add another feature that lets the user start the game again with a new random integer. Print an instruction like "TYPE 'YES' IF YOU WANT TO PLAY AGAIN." If he types 'YES' then start the game over; Otherwise, let the program stop.

TASK TWOS:

Write a program using a **FOR . . NEXT** loop to count by twos, up to a number typed by the user. If he types 8, your program should print
2
4
6
8

TASK BACK:

Use a **FOR . . NEXT** loop to count backwards from 20 to 0, by twos. You will need a STEP -2 in your 'FOR' statement.

41

## TASK NGREAT:

Ask the user to type two numbers, then compare them. If the user types 4 and 12.5, for example, your program should print

12.5 IS GREATER THAN 4

## TASK ALPH:

Compare two strings typed by the user. A string is "less than" another string if it comes before the other string alphabetically: "APPLE" < "FISH" is true. Your program should print something like

APPLE COMES BEFORE FISH

## TASK LLOOP:

Use a loop to get three numbers from the user, and print the largest of those numbers. Do not use three variables for the numbers. Hint: set a variable L (for largest) equal to 0. Then compare each user number with L. Change the value of L to a larger number if one is typed.

HINT:
Set a variable L (for largest) equal to zero. Then compare each user number with L. Change the value of L to a larger number if one is typed.

## TASK SLIST:

Let the user input a *list* of 4 strings (a *subscripted variable* with 4 "slots" in it) -- for example, the names of the courses he is taking. Print out the list after it is all typed in. uSe a **FOR . . NEXT** loop in this program.

HINT:
There are two parts to this:
Looping to input a string list, and looping to print it out.

SUB:
  Think about a number list for now. The key is to use
  the index of the loop as the index of the list. Write a
  loop whose index starts at 1 and goes to 4. The work of
  the loop is to assign the value of the index to the
  corresponding element of the list:
    L(I) = I
  The only way to test your program is to use another
  loop, indexed from 1 to 4, whose work is to print the
  list, one element at a time:
    PRINT L(I)
  The first execution of the loop should print the first
  element of the list, etc. When you finish this sub
  task, return to the main task. Change the list variable
  to a string list variable, and change the work of the
  first loop so that each execution asks the user to input
  a string.


TASK BACKLST:

  Take a list of strings from the user, then print the
  list in the opposite order. The list may be of any
  length up to 25 (ask how long the user wants it to be,
  then set up a loop whose top value is that number.) You
  will need a **FOR . . NEXT** loop with a STEP -1 to
  print the list backwards.

SUB:
  Very important:
  Write a loop that prints the value of its index. Start
  the loop at 1, but let the user give the top value. You
  can add to this program, making the loop do some real
  work, and the work will then be done as many times as
  the user likes.


TASK OTHER:

  Take a list of numbers from the user, of any length he
  likes up to 15. After he types the numbers, print out
  every other number in his list. (If he types these 6
  numbers: 2 8 12 5 3 9 your program should print the 2,
  12, and 3.)

HINT:
  Use a **FOR . . NEXT** loop with STEP 2. Then use the
  index of the loop as the index of the list to get every
  other element in the list.


43

## APPENDIX B

## THE BIP COMMANDS

This is an alphabetic listing of the BIP commands and their functions. Many (e.g., RUN, LIST, SAVE) are identical in function to their standard BASIC counterparts. The others serve specifically instructional purposes, in that they deal with BIP's curriculum structure, file system, or student history.

CALC
: Evaluates an expression. This feature allows the student to see the result of quick calculations without writing and running a complete program.

CURRIC
: Writes the text of the curriculum to a disk file. This is available to Stanford programmers and designated course instructors only. CURRIC provides a readable version of the curriculum-related parts of the network, with the text of the tasks listed along with the associated hints and subtasks. This listing appears as Appendix B.

ENOUGH
: Terminates the current task without giving the student credit for having completed it.

FILES
: Lists the names of the files in permanent storage with their last write dates.

FIX
: Allows the student to leave a message for Stanford.

FLOW        Generates and displays a flowchart
            representation of the student's current
            program. As the student steps through the
            execution, the element of the flowchart
            representing the current line blinks. This
            option is under development, and will be
            available only on CRT display terminals.

GET <name>  Retrieves the named program from permanent
            storage. The retrieved program replaces the
            current program (if any) in the student's
            core space.

HINT        Prints a hint, if any remain. Some tasks
            have more than one associated with them in
            the network; a few have no hints. When a
            student asks for a hint, BIP internally
            flags the hint that it supplies. Another
            request for a hint, during work on the same
            task, initiates a search for an associated
            hint not yet flagged.

KILL <name> Erases the named program from permanent
            storage. Students cannot affect each
            other's file storage, so indiscriminate use
            of this command can inconvenience only the
            KILLer himself.

LIST        Prints the current program in the order of
            its line numbers. Students are encouraged
            to LIST often, in order to avoid confusion
            between what was intended and what actually
            exists in the program.

MERGE <name> Retrieves the named program from permanent
            storage and adds it to the current program.
            Unlike GET, MERGE does not erase the current
            program before retrieval. MERGE allows the
            student to develop larger programs, a
            section at a time, testing and saving
            separate pieces the program as he goes. BIP
            informs him of instances in which a line
            from permanent storage replaces or
            duplicates the current line (i.e., where the
            two programs have one or more identically-
            numbered lines).

MODEL              Prints a typical solution to the current
                   task, only after all available hints and
                   subtasks have been presented. The student
                   may also request the model solution to a
                   task other than the current task by typing
                   its name as part of the MODEL command.

MORE               Continues the presentation of a task. If
                   all parts of the task have been completed,
                   the posttask interview is presented. Some
                   tasks require that the student complete two
                   or three closely related problems, calling
                   for a modification or expansion of the
                   original program. These "must-follow" tasks
                   are referred to as MORTs, both internally in
                   BIP and in the curriculum listing given in
                   Appendix B. The MORE routine will not allow
                   a student to advance, either to a MORT or to
                   a new task, unless he has successfully run
                   his current program.

REPORT             Provides Stanford programmers and designated
                   course instructors a summary of student
                   activity, either by school (currently DeAnza
                   or the University of San Francisco) or for
                   all students using BIP. The report shows
                   student number, name, number of sessions and
                   total hours accumulated on the course, and
                   number of tasks completed.

RES                Terminates all currently entered tasks,
                   without giving the student credit for
                   completing them. This option allows him to
                   extricate himself from a nest of tasks,
                   should the need arise.

RUN                Executes the current program.

SAVE <name>        Stores the current program for future use.
                   Saving the program in permanent storage does
                   not affect the current version in any way.

46

SCR            Erases the current program.

SIMPER         Allows the BIP student to use a simulated
               three-register machine described in Lorton &
               Slimick (1969). The SIMPER option allows
               instructors to demonstrate the differences
               between BASIC and a machine language by
               assigning problems to be solved with both.

SUB            Presents a subtask -- a smaller part needed
               to complete the current task at the
               student's request. Upon completion of a
               subtask, BIP returns the student
               automatically and explicitly to the larger
               task.

TASK <name>    Presents the student's next programming
               task. He may request a task of his choice
               by supplying its name; otherwise, BIP
               selects the next task on the basis of the
               student's history on previous tasks.

TRACE          Executes a program, but prints out line
               numbers and variables as execution
               progresses.

WHAT           Gives the name of the current task and
               (optionally) prints the problem text again.
               The student may request the text of a
               different task by supplying its name.

WHEN           Prints the current date and time.

WHO            Prints the name of the student signed on to
               the terminal. This option was included
               because of past experience with groups of
               students sharing a small number of
               terminals, and is intended to prevent the
               inadvertent termination of unfinished
               session.

# REFERENCES

Albrecht, R.L., Finkel, L., & Brown, J. R. BASIC, New York: Wiley, 1973.

Beard, M.H., Lorton, p., Jr., Searle, B. W., & Atkinson, R. C. Comparison of student performance and attitude under three lesson selection strategies in computer-assisted instruction, (Technical Report No. 222) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Carbonell, J. R. AI in CAI: An artificial intelligence approach to computer-assisted instruction. IEEE Transactions on Man-Machine Systems, 1970, MMS-11, 190-202.

Collins, A.M., Carbonell, J. R., & Warnock, E. H. Analysis and synthesis of tutorial dialogues. (Technical Report No. 2631) Cambridge, Mass.: Bolt, Beranek and Newman, 1973.

Coan, J.S. BASIC. New York: Hayden Book, 1970.

Feldman, J. A., Low, J. R., Swinehart, D. C., & Taylor, R. H. Recent developments in SAIL, AFIPS Fall Joint Conference, 1972, 1193-1202.

Floyd, R.W. Notes on programming and the ALGOL W language. Stanford, Calif.: Computer Science Department, Stanford University, 1971.

Forsythe, A.I., Keenan, T. A., Organick, E. I., & Sternberg, W. Computer science: A first course. New York: Wiley, 1969.

Friend, J. <u>Computer-assisted instruction in programming: A curriculum description</u>. (Technical Report No. 211). Stanford, Calif.: Institute for Mathetical Studies in the Social Sciences, Stanford University, 1973.

Goldberg, A. <u>Computer-assisted instruction: The application of theorem-proving to adaptive response analysis</u>. (Technical Report No. 203) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Kemeny, J. G. & Kurtz, T. E. BASIC <u>programming</u> (2nd ed) New York: Wiley, 1971.

Kimball, R. B. <u>Self-optimizing computer-assisted tutoring: Theory and practice</u>. (Technical Report No. 206) Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Koffman, E. B. & Blount, S. <u>A modular system for generative CAI in machine language programming</u>, Storrs, Conn.: University of Connecticut, School of Engineering, 1973.

Lorton, P., Jr. & Slimick, j. Computer based instruction in computer programming -- a symbol manipulation-list processing approach. <u>Proceedings of the Fall Joint Computer Conference</u>, 1969, 535-544.

Manna, Z. Program schemas. In A.V. Aho (Ed.), <u>Currents in the theory of computing</u>, Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Nievergelt, J., Reingold, E. M., & Wilcox, T. R. The automation of introductory computer science courses. <u>Proceedings of the International Computing Symposium</u>, 1973.

People's Computer Company Newsletter, Box 310, Menlo Park, Calif.

Nolan, R.L. <u>Introduction to computing through the BASIC language</u>. New York: Holt, Rinehart and Winston, 1969.

Smith, R. TENEX SAIL. Technical Report   in   preparation. Stanford, Calif.:  Institute for  Mathematical Studies in the Social Sciences, Stanford University, 1974.

Swinehart, D. C.,  & Sproull, R. F.  SAIL,  Stanford, Calif: Stanford Artificial Intelligence  Laboratory Operating Note 57.2, Stanford University, 1971.

VanLehn,  K.,  SAIL  User  Manual,  Stanford,  Calif: Stanford Artificial    Intelligence    Laboratory,    Stanford University, 1973.

Wiener,  H.,  &  Ross, B.   BASIC workbook.   Berkeley, Calif.: Lawrence  Hall of  Science, University  of California, 1972.